Using Gloo as an Ingress Gateway for AWS App Mesh

Frédéric Médery, Solutions Architect

As part of their organization’s digital transformation, more and more customers are electing to use a managed Kubernetes service, like *Amazon EKS*,. as their container-orchestration system of choice to deploy, scale, and manage microservices.

As the number of microservices grow within an application, it becomes difficult to pinpoint the exact location of errors, re-route traffic after failures, and safely deploy code changes. A service mesh, like *AWS App Mesh*, makes it easy to run services by providing consistent visibility and network traffic controls for services built across multiple types of compute infrastructure such as Amazon EC2, Amazon EKS and AWS Fargate.

Service mesh is great to handle service-to-service communication, but developers still need to create and manage access to their applications from outside the Kubernetes cluster. Ingress controller has become a very popular solution to solve this type of challenge.

Today, we will take a look at Gloo (from solo.io) which is a feature-rich, Kubernetes-native ingress controller, and API gateway based on Envoy. Gloo is exceptional in its function-level routing, its support for legacy apps, microservices, and serverless applications.   It can also be easily integrated with *AWS App Mesh.*

In this blog post:

- We will create an EKS cluster with add-ons to create and manage AWS App Mesh automatically.
- We will deploy a sample application integrated with AWS App Mesh.
- We will install and configure Gloo as the Ingress Controller.
- And finally we will use Gloo for a canary deployment.

Prerequisites

Before starting, we need to install the following tools on our local computer:

- kubectl 1.13
- awscli (installed and configured)
- aws-iam-authentificator
- eksctl
- glooctl

Getting started

We will start by creating an EKS cluster in the `us-east-2` region using a yaml config file:

```
#create eks config file
cat <<EoF > eks-config.yaml
```

```
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: GlooDemo
  region: us-east-2
  version: "1.13"

nodeGroups:
  - name: ng1-GlooDemo
    instanceType: m5.large
    desiredCapacity: 2
    iam:
      withAddonPolicies:
        autoScaler: true
        #allow access AWS app mesh
        appMesh: true
        #allow access to AWS X-Ray
        xRay: true
        #allow access to AWS CloudWatch
        cloudWatch: true
EoF

#create eks cluster
eksctl create cluster --auto-kubeconfig -f eks-config.yaml

#load the EKS cluster config
export KUBECONFIG=${HOME}/.kube/eksctl/clusters/GlooDemo
```

The cluster creation will take up to 15 minutes.

To simplify the creation and the management of *AWS App Mesh* we will install 2 add-ons:

- **aws-app-mesh-controller-for-k8s** that will manage *AWS App Mesh* resources for a Kubernetes cluster.
- **aws-app-mesh-inject** that will be responsible for automatically inject the app mesh container as a sidecar. To enable sidecar injection for a namespace, it is necessary to label the namespace with `appmesh.k8s.aws/sidecarInjectorWebhook=enabled`

```
#install aws-app-mesh-controller-for-k8s
kubectl apply -f https://raw.githubusercontent.com/aws/aws-app-mesh-controller-for-k8s
```

Now we will confirm that the Kubernetes custom resources for `AWS App Mesh` were created with the following command:

```
kubectl get crd

NAME                                  CREATED AT
eniconfigs.crd.k8s.amazonaws.com   2019-09-19T23:48:42Z
meshes.appmesh.k8s.aws             2019-09-20T14:00:46Z
virtualnodes.appmesh.k8s.aws       2019-09-20T14:00:46Z
virtualservices.appmesh.k8s.aws    2019-09-20T14:00:46Z
```

The `aws-app-mesh-inject` add-on needs to know the name of the mesh before being installed.
We will use `color-mesh` as our mesh name, and to help with visibility and tracing, we will also add the X-Ray container as a sidecar:

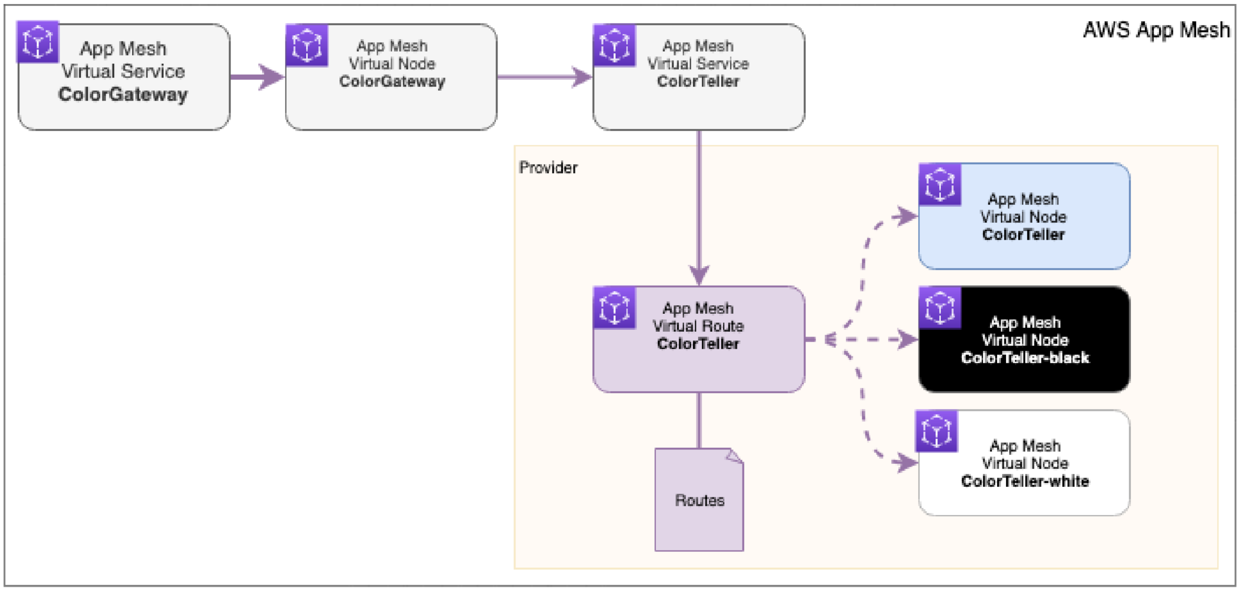
```
#export mesh name
export MESH_NAME="color-mesh"

#install X-Ray sidecar
export INJECT_XRAY_SIDECAR="true"
export ENABLE_STATS_TAGS="true"
export ENABLE_STATSD="true"

#install the sidecar injector
curl https://raw.githubusercontent.com/aws/aws-app-mesh-inject/master/scripts/install.
```

Deploying a mesh connected sample application

The sample application consists of two components:

- **ColorGateway** – A simple http service written in Go that is exposed to external clients and that responds to *http://service-name:port/color*. The gateway responds with a color retrieved from *color-teller* and a histogram of colors observed at the server that responded up to the point when you made the request.
- **ColorTeller** – A simple http service written in Go that is configured to return a color. Multiple variants of the service are deployed. Each service is configured to return a specific color.

ColorGateway isn't aware of the multiple variants of the CollorTeller component. *AWS App Mesh* will expose one CollorTeller virtual service with 3 virtual nodes and 3 routes.



To deploy the sample application, apply the following file to your Kubernetes cluster with the following command.

```
#install sample application
kubectl apply -f https://raw.githubusercontent.com/fmedery/blogs/master/eks-appmesh-gl
```

You can verify all the deployed objects using the following command (notice that 2 extra containers are running in each pods: one for AWS X-Ray and one for AWS App Mesh.

```
kubectl -n appmesh-demo get all
NAME                                     READY   STATUS    RESTARTS   AGE
pod/colorgateway-957f89b6b-d6m9j         3/3     Running   0          4m34s
pod/colorteller-759fc757cc-lcmvz         3/3     Running   0          4m34s
pod/colorteller-black-6c5dd7689c-sp9ng   3/3     Running   0          4m33s
pod/colorteller-blue-58dbf546d5-lfg4j    3/3     Running   0          4m33s

NAME                          TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
service/colorgateway          ClusterIP   10.100.77.66     <none>        9080/TCP   4m34
service/colorteller           ClusterIP   10.100.48.110    <none>        9080/TCP   4m34
service/colorteller-black     ClusterIP   10.100.65.11     <none>        9080/TCP   4m33
service/colorteller-blue      ClusterIP   10.100.173.174   <none>        9080/TCP   4m33

NAME                                 READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/colorgateway         1/1     1            1           4m34s
deployment.apps/colorteller          1/1     1            1           4m34s
deployment.apps/colorteller-black    1/1     1            1           4m33s
deployment.apps/colorteller-blue     1/1     1            1           4m33s

NAME                                            DESIRED   CURRENT   READY   AGE
replicaset.apps/colorgateway-957f89b6b          1         1         1       4m34s
replicaset.apps/colorteller-759fc757cc          1         1         1       4m34s
replicaset.apps/colorteller-black-6c5dd7689c    1         1         1       4m33s
replicaset.apps/colorteller-blue-58dbf546d5     1         1         1       4m33s

NAME                                                   AGE
virtualservice.appmesh.k8s.aws/colorgateway.appmesh-demo   4m
virtualservice.appmesh.k8s.aws/colorteller.appmesh-demo    4m

NAME                                 AGE
mesh.appmesh.k8s.aws/color-mesh      4m

NAME                                           AGE
virtualnode.appmesh.k8s.aws/colorgateway       4m
virtualnode.appmesh.k8s.aws/colorteller        4m
virtualnode.appmesh.k8s.aws/colorteller-black  4m
virtualnode.appmesh.k8s.aws/colorteller-blue   4m
```

Installing Gloo

Now that our sample application has been installed and integrated with AWS App Mesh, we will install `Gloo` ingress gateway.

```
#install gloo gateway using the gloo command line interface
glooctl install gateway
```

We can get an overview of all of the resources running using the command below (notice that `Gloo` automatically provisioned an `AWS Elastic Load Balancer`).

```
kubectl get all -n gloo-system
NAME                                    READY   STATUS    RESTARTS   AGE
pod/discovery-85df7bd6db-287kv          1/1     Running   0          4m28s
pod/gateway-proxy-v2-5b55756ddc-cbkmq   1/1     Running   0          4m28s
pod/gateway-v2-7db445675d-v296x         1/1     Running   0          4m28s
pod/gloo-9f499868d-fx8w5                1/1     Running   0          4m28s

NAME                       TYPE           CLUSTER-IP       EXTERNAL-IP
service/gateway-proxy-v2   LoadBalancer   10.100.255.203   a859df235dbaf11e982e0027727
service/gloo               ClusterIP      10.100.97.202    <none>

NAME                               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/discovery          1/1     1            1           4m28s
deployment.apps/gateway-proxy-v2   1/1     1            1           4m28s
deployment.apps/gateway-v2         1/1     1            1           4m28s
deployment.apps/gloo               1/1     1            1           4m28s

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/discovery-85df7bd6db          1         1         1       4m28s
replicaset.apps/gateway-proxy-v2-5b55756ddc   1         1         1       4m28s
replicaset.apps/gateway-v2-7db445675d         1         1         1       4m28s
replicaset.apps/gloo-9f499868d                1         1         1       4m28s

NAME                             AGE
mesh.appmesh.k8s.aws/color-mesh   5m
```

Now, let's have a look at the pods:

- `pod/discovery-85df7bd6db-287kv`: this component is responsible for dynamically discovering services to which Gloo can route (upstreams).
- `pod/gateway-proxy-v2-5b55756ddc-cbkmq`: this is the `Envoy` proxy
- `pod/gateway-v2-7db445675d-v296x`: this component allows users to configure an `Envoy` Proxy and also generates configuration that the `Gloo` control plane can use to generate `Envoy` configuration through xDS
- `pod/gloo-9f499868d-fx8w5`: an event-driven component responsible for generating configuration for and serving the core xDS services and configuration of custom Envoy filters.

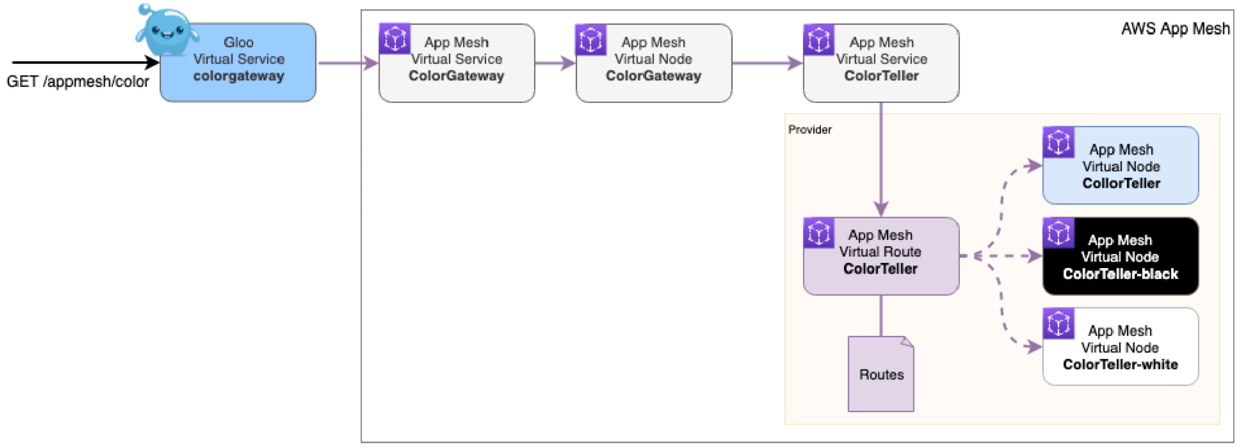
Because of these decoupling developers contributing to `Gloo` can easily add support for other architectures like `Knative` for example.

Allowing ingress connectivity using Gloo

Let's dig a little deeper into two of Gloo core concepts `Virtual Service` and `Upstream`:

- **Virtual Service** defines a set of route rules that live under a domain or set of domains. Route rules consist of a matcher which specifies the kind of function calls to match (requests and events, are currently supported), and the name of the destination (or destinations) to route them to.

- **Upstream** defines destinations for routes. Upstreams tell `Gloo` what to route to. Upstreams are automatically discovered by Gloo.


GET /appmesh/color
Gloo Virtual Service colorgateway
AWS App Mesh
App Mesh Virtual Service ColorGateway
App Mesh Virtual Node ColorGateway
App Mesh Virtual Service ColorTeller
Provider
App Mesh Virtual Route ColorTeller
Routes
App Mesh Virtual Node CollorTeller
App Mesh Virtual Node ColorTeller-black
App Mesh Virtual Node ColorTeller-white

We will first verify that the ColorGateway virtual node was discovered by Gloo.

```
glooctl get upstreams|grep colorgateway

| appmesh-demo-colorgateway-9080    | Kubernetes | Accepted | svc name: colorgateway |
| appmesh-demo-colorgateway-v1-9080 | Kubernetes | Accepted | svc name: colorgateway |
```

We will create a `Gloo Virtual Service` called `colorgateway` and a route that will redirect the path `/appmesh/color` to the virtual service `appmesh-demo-colorgateway-9080` using `/color` as his path.

```
#create a virtual service called colorgateway and a route to reach it
glooctl add route \
            --name colorgateway \
            --path-prefix /appmesh/color \
            --prefix-rewrite /color \
            --dest-name appmesh-demo-colorgateway-9080
+-----------------+--------------+---------+------+----------+-----------------+------
| VIRTUAL SERVICE | DISPLAY NAME | DOMAINS | SSL  |  STATUS  | LISTENERPLUGINS |
+-----------------+--------------+---------+------+----------+-----------------+------
| colorgateway    | colorgateway | *       | none | Accepted |                 | /appm
|                 |              |         |      |          |                 | gloo-
|                 |              |         |      |          |                 | (upst
+-----------------+--------------+---------+------+----------+-----------------+------
```

Now we will try to connect to the application.

```
#verify that the load balancer URL exists
glooctl proxy url
```

```
http://aa1706959c82111e9bdfc02f316f0629-1493319117.us-east-2.elb.amazonaws.com:80

#test the connectivity
for i in {1..10}; do curl $(glooctl proxy url)/appmesh/color; echo; done
{"color":"blue", "stats": {"black":0.3,"blue":0.13,"white":0.57}}
{"color":"white", "stats": {"black":0.29,"blue":0.13,"white":0.58}}
{"color":"white", "stats": {"black":0.28,"blue":0.12,"white":0.6}}
{"color":"black", "stats": {"black":0.31,"blue":0.12,"white":0.58}}
{"color":"black", "stats": {"black":0.33,"blue":0.11,"white":0.56}}
{"color":"white", "stats": {"black":0.32,"blue":0.11,"white":0.57}}
{"color":"blue", "stats": {"black":0.31,"blue":0.14,"white":0.55}}
{"color":"blue", "stats": {"black":0.3,"blue":0.17,"white":0.53}}
{"color":"black", "stats": {"black":0.32,"blue":0.16,"white":0.52}}
{"color":"black", "stats": {"black":0.34,"blue":0.16,"white":0.5}}
```

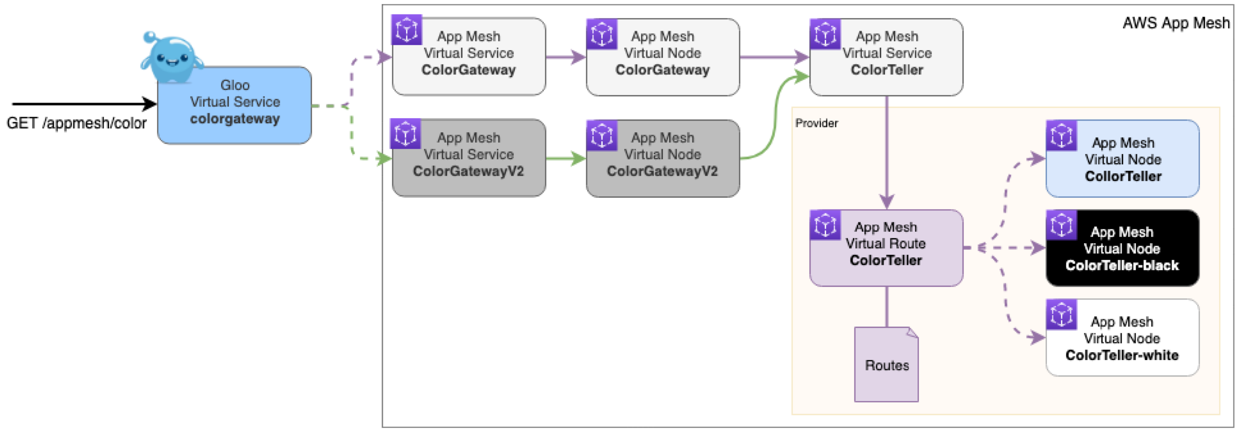
As we can see, we now have a route `/appmesh/color` that redirects the traffic to the *AWS App Mesh* virtual service `ColorGateway`.

Canary Deployment using Gloo weighted destinations

With this section, we will introduce another important concept of Gloo, `Upstream Group`:

- **Upstream Group** is a top-level object, that let you logically groups `upstreams`, giving you the ability to address them as a group in distinct VirtualServices

This is a common requirement for Canary deployments where you want all calling routes to forward traffic equally across the two service versions.



For this example, we will deploy a new version of the gateway called `colorteller-gatewayV2` and an `upstream group` with both version of the gateway.

```
#deploy colorteller-gatewayV2
kubectl apply -f https://raw.githubusercontent.com/fmedery/blogs/master/eks-appmesh-g

#verify if Gloo was able to discover it
glooctl get upstream | grep gatewayv2
```

```
|appmesh-demo-colorgatewayv2-9080   | Kubernetes| Accepted| svc name: colorgatewayv2|
|appmesh-demo-colorgatewayv2-v2-9080| Kubernetes| Accepted| svc name: colorgatewayv2|
```

Now we will create an `upstream group` called `upstreamgroup-gateway` and add routes that will split the traffic between `colorteller-gateway` (80%) and `colorteller-gatewayv2` (20%).

```
#delete the virtual service previously created
glooctl delete vs --name colorgateway

#re create the colorgateway virtual service with an upstream group has its backend
kubectl apply -f https://raw.githubusercontent.com/fmedery/blogs/master/eks-appmesh-g

upstreamgroup.gloo.solo.io/upstreamgroup-gateway created
virtualservice.gateway.solo.io/colorgateway configured

#verify the upstreamgroup has been created
glooctl get upstreamgroup --name upstreamgroup-gateway
+-----------------------+----------+--------------+----------------------------------+
|    UPSTREAM GROUP     |  STATUS  | TOTAL WEIGHT |             DETAILS              |
+-----------------------+----------+--------------+----------------------------------+
| upstreamgroup-gateway | Accepted | 100          | destination type: Upstream       |
|                       |          |              | namespace: gloo-system           |
|                       |          |              | name:                            |
|                       |          |              | appmesh-demo-colorgateway-9080   |
|                       |          |              | weight: 80   % total: 0.80       |
|                       |          |              |                                  |
|                       |          |              |                                  |
|                       |          |              | destination type: Upstream       |
|                       |          |              | namespace: gloo-system           |
|                       |          |              | name:                            |
|                       |          |              | appmesh-demo-colorgatewayv2-9080 |
|                       |          |              | weight: 20   % total: 0.20       |
+-----------------------+----------+--------------+----------------------------------+

#verify that we have a route pointing to the upstreamgroup
 glooctl get vs --name colorgateway
+-----------------+--------------+---------+------+----------+-----------------+------
| VIRTUAL SERVICE | DISPLAY NAME | DOMAINS | SSL  |  STATUS  | LISTENERPLUGINS |
+-----------------+--------------+---------+------+----------+-----------------+------
| colorgateway    |              | *       | none | Accepted |                 | /appm
|                 |              |         |      |          |                 | upstr
+-----------------+--------------+---------+------+----------+-----------------+------
#test the new route
for i in {1..10}; do curl $(glooctl proxy url)/appmesh/color; echo; donex`
{"color":"white", "stats": {"black":0.32,"blue":0.35,"white":0.33}}
{"color":"white", "stats": {"black":0.31,"blue":0.34,"white":0.34}}
{"color":"blue", "stats": {"black":0.31,"blue":0.36,"white":0.34}}
{"colorV2":"white", "statsV2": {"black":0.24,"blue":0.57,"white":0.19}}
{"color":"blue", "stats": {"black":0.3,"blue":0.37,"white":0.33}}
{"color":"black", "stats": {"black":0.31,"blue":0.36,"white":0.33}}
{"color":"blue", "stats": {"black":0.31,"blue":0.37,"white":0.32}}
{"color":"black", "stats": {"black":0.32,"blue":0.37,"white":0.32}}
```

```
{"colorV2":"blue", "statsV2": {"black":0.23,"blue":0.59,"white":0.18}}
{"color":"blue", "stats": {"black":0.31,"blue":0.38,"white":0.31}}
```

When we are confident that the new version is behaving as expected we can increase the traffic sent to `colorteller-gatewayV2`.

```
#update the weight of each route to 50%
kubectl apply -f https://raw.githubusercontent.com/fmedery/blogs/master/eks-appmesh-gl
upstreamgroup.gloo.solo.io/upstreamgroup-gateway configured

#verify the weight for each route has been updated to 50%
glooctl get upstreamgroup --name upstreamgroup-gateway
+-----------------------+----------+--------------+----------------------------------+
|    UPSTREAM GROUP     |  STATUS  | TOTAL WEIGHT |             DETAILS              |
+-----------------------+----------+--------------+----------------------------------+
| upstreamgroup-gateway | Accepted | 100          | destination type: Upstream       |
|                       |          |              | namespace: gloo-system           |
|                       |          |              | name:                            |
|                       |          |              | appmesh-demo-colorgateway-9080   |
|                       |          |              | weight: 50   % total: 0.50       |
|                       |          |              |                                  |
|                       |          |              |                                  |
|                       |          |              | destination type: Upstream       |
|                       |          |              | namespace: gloo-system           |
|                       |          |              | name:                            |
|                       |          |              | appmesh-demo-colorgatewayv2-9080 |
|                       |          |              | weight: 50   % total: 0.50       |
+-----------------------+----------+--------------+----------------------------------+

#test the route again
for i in {1..10}; do curl $(glooctl proxy url)/appmesh/color; echo; done
{"colorV2":"white", "statsV2": {"black":0.22,"blue":0.57,"white":0.22}}
{"color":"blue", "stats": {"black":0.31,"blue":0.38,"white":0.31}}
{"color":"white", "stats": {"black":0.3,"blue":0.38,"white":0.32}}
{"colorV2":"white", "statsV2": {"black":0.21,"blue":0.54,"white":0.25}}
{"colorV2":"blue", "statsV2": {"black":0.2,"blue":0.56,"white":0.24}}
{"color":"black", "stats": {"black":0.31,"blue":0.37,"white":0.31}}
{"colorV2":"white", "statsV2": {"black":0.19,"blue":0.54,"white":0.27}}
{"colorV2":"white", "statsV2": {"black":0.19,"blue":0.52,"white":0.3}}
{"colorV2":"black", "statsV2": {"black":0.21,"blue":0.5,"white":0.29}}
{"color":"black", "stats": {"black":0.32,"blue":0.37,"white":0.31}}
```

Now we will route all the traffic to `colorteller-gatewayv2` by removing `colorteller-gateway` from the upstream group:

```
# update the upstream group
kubectl apply -f https://raw.githubusercontent.com/fmedery/blogs/master/eks-appmesh-gl

#verify the upstreamgroup
glooctl get upstreamgroup --name upstreamgroup-gateway
+-----------------------+----------+--------------+-----------------------------------
|    UPSTREAM GROUP     |  STATUS  | TOTAL WEIGHT |               DETAILS
```

```
+-----------------------+----------+--------------+-----------------------------------
| upstreamgroup-gateway | Accepted | 100          | destination type: Upstream
|                       |          |              | namespace: gloo-system
|                       |          |              | name:
|                       |          |              | appmesh-demo-colorgatewayv2-9080
|                       |          |              | weight: 100   % total: 1.00
+-----------------------+----------+--------------+-----------------------------------

#test one last time
for i in {1..10}; do curl $(glooctl proxy url)/appmesh/color; echo; done
{"colorV2":"black", "statsV2": {"black":0.24,"blue":0.48,"white":0.28}}
{"colorV2":"blue", "statsV2": {"black":0.23,"blue":0.5,"white":0.27}}
{"colorV2":"white", "statsV2": {"black":0.23,"blue":0.48,"white":0.29}}
{"colorV2":"blue", "statsV2": {"black":0.22,"blue":0.5,"white":0.28}}
{"colorV2":"black", "statsV2": {"black":0.24,"blue":0.48,"white":0.27}}
{"colorV2":"white", "statsV2": {"black":0.24,"blue":0.47,"white":0.29}}
{"colorV2":"white", "statsV2": {"black":0.23,"blue":0.46,"white":0.31}}
{"colorV2":"black", "statsV2": {"black":0.25,"blue":0.44,"white":0.31}}
{"colorV2":"blue", "statsV2": {"black":0.24,"blue":0.46,"white":0.3}}
{"colorV2":"blue", "statsV2": {"black":0.24,"blue":0.47,"white":0.29}}
```

Gloo is now routing 100% of the traffic hitting `/appmesh/color` to `colorteller-gatewayv2`.

Conclusion

In this article we demonstrated how `Gloo` can transparently interact with `AWS App Mesh` and how easily developers can create and manipulate access to their applications. We also showcase how to use `Gloo` to create a canary deployment.

To learn more about Gloo advanced features follow this link.

In a next article we will take a look at another solo.io product: supergloo, *the Service Mesh Orchestration Platform.*